Colecciones

Son estructuras de datos que nos permiten guardar en su interior cualquier tipo de información. La gran diferencia entre las colecciones y los arreglos, es que en las primeras se pueden apregar elementos dinámicamente, sin la necesidad de establecer el tamaño (cantidad de elementos) previamente como lo hacemos en los arreglos.

Importante: para poder usar colecciones es necesario que nuestro programa utilice el namespace "System.Collections".

(1) ArrayList:

es una lista de elementos de tamaño dinámico. La lista puede almacenar distintos tipos de valores (int, string, float, etc.).

(a) Declaración:

```
ArrayList datos = new ArrayList ();
```

Nota: es posible instanciar el arraylist con algún valor de capacidad inicial. Esto es útil cuando conocemos de antemano la cantidad de elementos que puede contener la lista (aunque siempre podrá crecer si fuese necesario).

```
ArrayList datos = new ArrayList (32);
```

(b) Adición de datos

```
datos.Add (12);
datos.Add ("Hola");
```

Nota: el método Add (dato a agregar); simplemente agrega al final de la lista el dato que le pasemos por parámetro.

(c) Inserción de datos

Sabemos que con Add(); podemos agregar un dato al final d la lista, pero ¿Si queremos insertar un dato en una posición específica?

```
datos.Insert (2, "Mundo");
```

→ El primer parámetro es el índice y el segundo es el dato a insertar.

(d) Eliminación de datos

datos.RemoveAt(7); → Eliminamos el dato que tenga índice 7.

(e) Acceso a los datos

Console.WriteLine("El valor es {0}", datos[2]);
↳ índice

(f) Modificación de datos

Si queremos reasignar un valor de un índice determinado, hacemos:

datos[2] = 5;
↳ índice ↳ Nuevo valor

(g) Conocer la cantidad de elementos del ArrayList.

int cantidad = 0;

cantidad = datos.Count;

(h) Conocer la capacidad

int capacidad = datos.Capacity; { Nos permite conocer la capacidad asignada. Si es cero, irá aumentando de 4 en 4 a medida que vamos agregando elementos

Nota: como la capacidad aumenta de 4 en 4, podemos establecerla en el número real de elementos que hay en la lista, mediante: datos.TrimToSize();

(i) Para buscar un valor dentro de la lista

int indice = datos.IndexOf("pepe"); } ¿Y si el valor no existe? devuelve -1

(j) Para limpiar el arraylist

datos.Clear(); → Quita todos los elementos del arraylist. No cambia su capacidad.

(K) Recorrer un ArrayList

Hay 2 estructuras repetitivas que nos permiten recorrer un ArrayList.

- FOREACH → La vamos a usar cuando los elementos del ArrayList son del mismo tipo.

```
foreach (string valor in datos)
{
    Console.WriteLine("El valor es: " + valor);
}
```

En este ejemplo son todos de tipo string.

Clase 8 — Colecciones
Página (3) — 213 a 253

- **FOR** → lo usaremos cuando los elementos del arraylist son de distinto tipo.

```
int it = 0;
for (it = 0; it < datos.Count; it++)
{
  Console.WriteLine ("El valor es: " + datos[it]);
}
```

Va a mostrar los valores aunque sean de distinto tipo

② El stack o "pila"

Es una colección de tipo LIFO (Last in, First out).

```
Stack pila = new Stack();
```

ⓐ Push (Apregar datos)

Permite apregar datos al final de la pila.

```
pila.Push (7);
pila.Push (11);
pila.Push ("hola");
```

ⓑ Pop (Extraer datos): Extraemos el dato que está al final de la pila (el dato se elimina de la pila).

```
int valor = 0;
valor = (int)pila.Pop();
```

si en la pila sólo tuviésemos valores enteros

```
var valor = pila.Pop();
```

si en la pila tenemos distintos tipos de datos

↳ El compilador determina qué tipo de dato hay almacenado en la variable.

ⓒ Peek (Leer): Devuelve el objeto situado al final de la pila sin eliminarlo.

```
var valor = pila.Peek();
```

ⓓ Recorrer un stack

```
foreach (var valor in pila)
{
    Console.WriteLine ("El valor es: " + valor);
}
```

El uso de var hace que el compilador determine el tipo de dato contenido en la posición de la pila.

(e) Count;

Obtiene la cantidad total de elementos de la pila

```
int cant = pila.Count;
```

(f) Clear();

Elimina todos los elementos de la fila.

```
pila.clear();
```

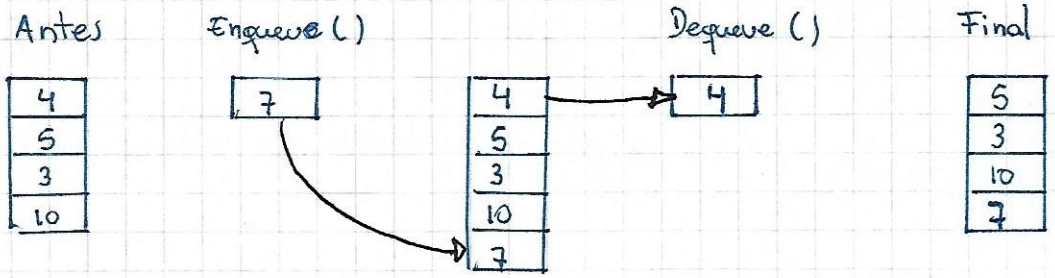
(g) Contains()

Permite buscar un elemento indicando su valor.

```
bool encontrado = false;
encontrado = pila.Contains("hola");
```

(3) La "cola" o Queue

A diferencia de la pila, es una estructura de tipo FiFo (First in, First out).



(a) Declarar la cola

```
Queue fila = new Queue();
```

(b) Agregar elemento

```
fila.Enqueue(7);
```
→ Lo agregamos al final de la cola

(c) Extraer elemento

```
var valor;
valor = fila.Dequeue();
```
→ Extrae el primer elemento de la cola. La cola quedará sin ese elemento.

(d) Contains()

```
bool encontrado = fila.Contains("Hola");
```
→ devuelve true o false

ⓔ Clear();

fila.Clear(); → borra todo el contenido de la cola

ⓕ Count;

int x=fila.Count; → devuelve la cant. de elementos de la cola

ⓖ Peek();

var dato = fila.Peek(); → Obtiene el 1er elemento sin eliminarlo de la Cola.

ⓗ Recorrer una Cola

```
foreach (var dato in fila)
{
    Console.WriteLine("El valor es: " + dato);
}
```

④ El Hashtable

Es una colección indexada. Es decir que tenemos un índice y un valor referenciado a ese índice.

En el Hashtable siempre utilizaremos una pareja de Key y Value.

ⓐ Declaración

```
Hashtable mitabla = new Hashtable();
```

ⓑ Agregar elementos con .Add()

```
mitabla.Add("Pan", 5.77);
mitabla.Add("Soda", 10.85);
mitabla.Add("Atun", 15.50);
```

ⓒ Leer un elemento

```
var valor = mitabla["Pan"];
```

} Podemos especificar el tipo si lo conocemos:
float valor = (float) mitabla["Pan"];

ⓓ Eliminar un elemento con .Remove(Key)

```
mitabla.Remove("Pan");
```

ⓔ ContainsValue (valor a buscar);

```
bool existe = mitabla.ContainsValue(17.50);
```

} Permite saber si existe ese valor en particular. Devuelve true o false.

(f) Contains (clave a buscar);

```
bool existe = mitabla.Contains ("Pan");
```

} Permite saber si existe la Key en el Hashtable, pero no devuelve su valor, sino un true o false.

(g) Para limpiar el Hashtable

```
mitabla.Clear();
```

(h) Count

```
int cant = mitabla.Count;
```

} devuelve la cantidad de elementos

(i) Recorrer el Hashtable

```
foreach (DictionaryEntry dato in mitabla) // la clase DictionaryEntry también guarda parejas Key-Value y nos permite iterar.
{
    Console.WriteLine ("Key: {0}, Value: {1}", dato.Key, dato.Value);
}
```

(j) Copiar sólo los valores en una colección

```
ICollection valores = mitabla.Values;
foreach (var valor in valores)
{
    Console.WriteLine ("El valor es: " + valor);
}
```

} ICollection es una interfase usada para implementar las colecciones, de tal forma que "valores" puede actuar como cualquier colección válida, por lo que también podemos recorrerla con un foreach.